Lecture 15

Debugging Code

MA 350

12 March 2020

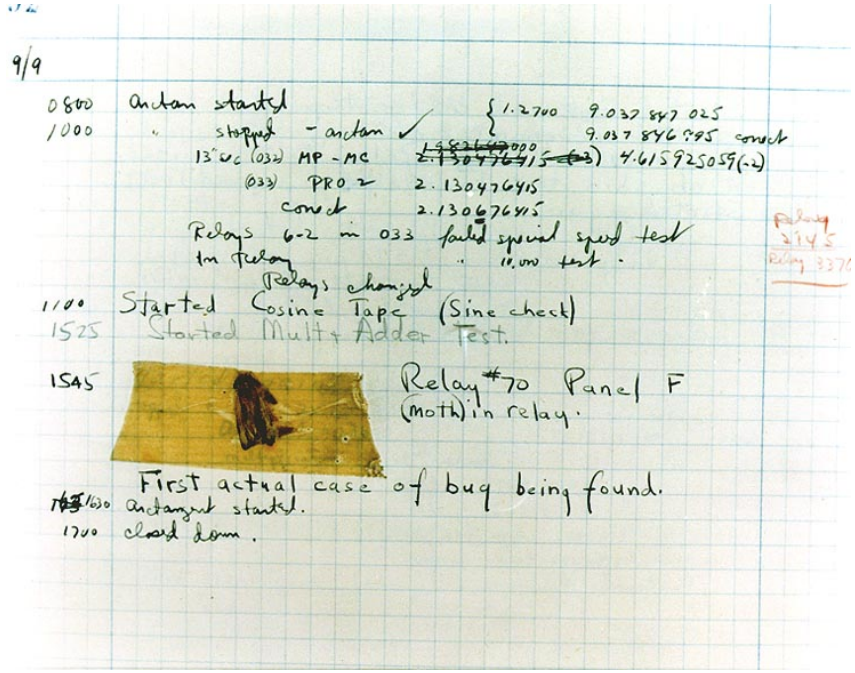
Debugging Basics

Bug!

The original name for glitches and unexpected defects:

- dates back at least to Thomas Edison in 1876.

A better story from Grace Hopper in 1947:



Debugging: What and Why?

Debugging is a the process of locating, understanding, and removing bugs from your code.

Why should we care to learn about this?

Short Answer:

- You will **always** have to debug your code, because you're not perfect (none of us are!) and can't write perfect code.

Longer Answer:

- Debugging is frustrating and time-consuming, but essential.
- Writing code that is easier to debug later is worth it, even if it takes a bit more time.
- Lots of our best-practices already help with debugging.
  - Use lots of comments, write modular code, use meaningful variable names, etc.

Common Bugs in R

Syntax Errors

A **syntax error** is when you write R code that will not run, since it does not parse-out to a valid R expression.

Common Syntax Errors:

- Using operators with the wrong data structures
  - Using `-` to negate a list.
  - Using `+` to add two lists, rather than two elements of a list.
- Unmatched parenthesis.
- Unmatched `{` or `}`.
- Just plain-wrong syntax:
  - `if()`, `for()`, `while()`, `function()`, etc.

Semantic Errors

A **semantic error** is when you write technically valid R code, but the code does not do what you want it to do.

Common Semantic Errors:

- Parenthesis missing or mis-matched
- `[[ ... ]]` vs. `[ ... ]`
- `==` vs. `=`
- Vectors vs. single values: code works for one value but not multiple ones, unexpected recycling
- Silent type conversions (`double` to `int`, etc.)

Semantic Errors Related to Variables and Functions

- Using the wrong variable(s).
  - Try to avoid single-letter names.
  - Try to avoid names like `var`, `list`, `vector`, etc.
- Using the wrong function(s).
  - Make sure you are sure what a function does before using it.
  - When in doubt, use `?`
  - Try the function out with a few inputs where you know what you expect as outputs.
- Giving unnamed arguments to a function in the wrong order.
- The R expression does not match the math you mean:
  - you left something out; you added something; you mistyped something

Semantic Errors Related to Scope and Global Variables

- Relying on a global variable which doesn't have the right value.
  - Or only has the right value in *one* situation but not another.
- Assuming that changing a variable inside the function will change it elsewhere.
- Confusing variables within a function and those from where the function was called.

How to Debug in R

Debugging: How?

Debugging is (largely) a process of differential diagnosis.

> All happy families are alike; each unhappy family is unhappy in its own way. — Leo Tolstoy, *Anna Karenina*

Debugging: How?

Stages of Debugging

0. **Reproduce the Error:** can you make the bug reappear?
1. **Characterize the Error:** what can you see that is going wrong?
2. **Localize the Error:** where in the code does the mistake originate?
3. **Modify the Code:** did you eliminate the error? Did you add new ones?

Reproduce the Bug

Step 0: Make It Happen Again

- Can we produce it repeatedly when re-running the same code, with the same input values?
- And if we run the same code in a clean copy of R, does the same thing happen?

Characterize the Bug

Step 1: Figure Out If It's a Pervasive / Big Problem

- How much can we change the inputs and get the same error?
- Or do we get a different error?
- And how big is the error?

Localize the Bug

Step 2: Find Out Exactly Where Things Are Going Wrong

- This is most often the hardest part!
- Investigate errors, using `traceback()`, `cat()`, and `print()`
- Interactively debug with `browser()`.

Localizing Can Be Easy or Hard

Sometimes error messages are easier to decode, sometimes they're harder.

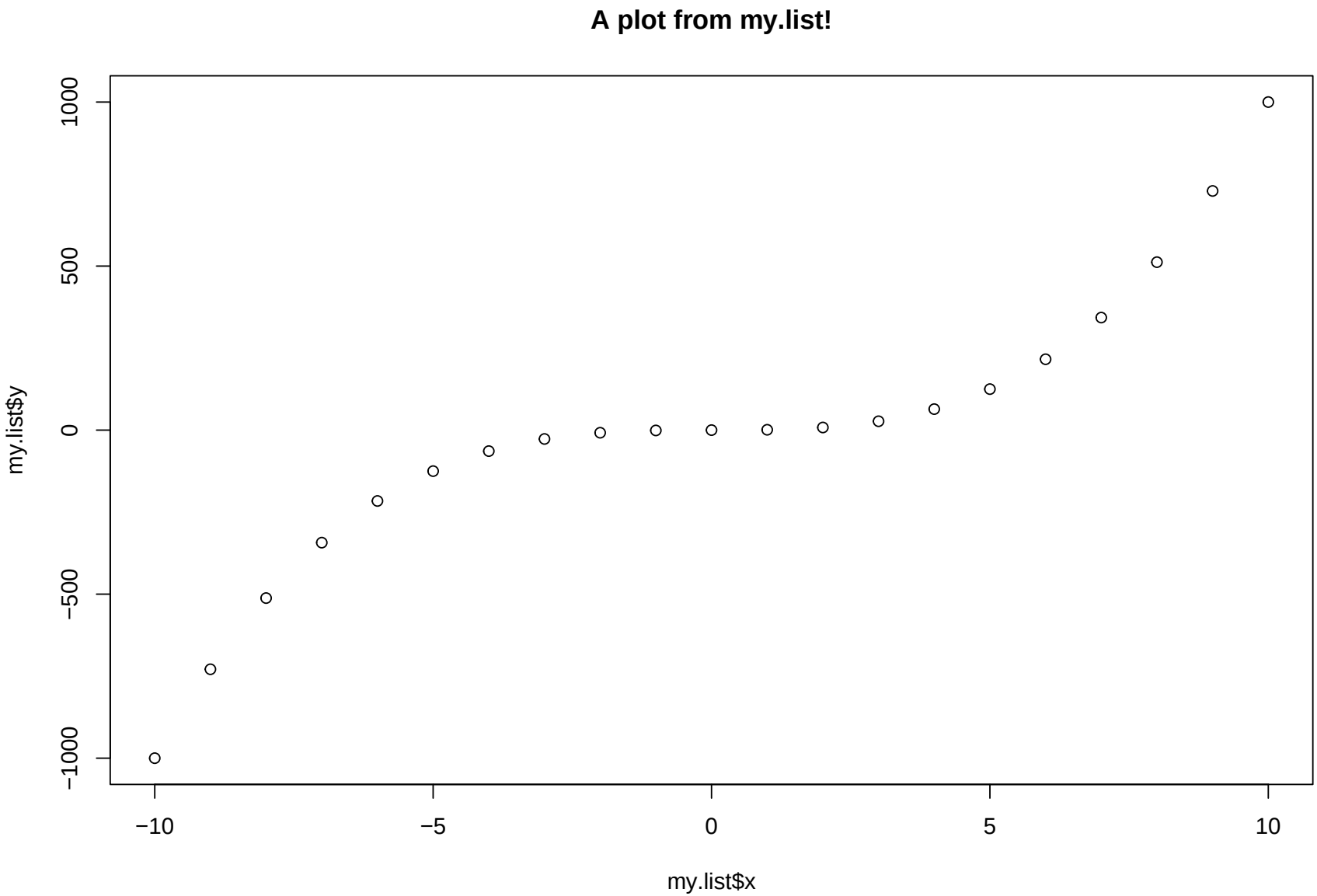
This can make locating the bug easier or harder.

Example:

```
my.plotter = function(x, y, my.list=NULL) {
  if (!is.null(my.list))
    plot(my.list, main="A plot from my.list!")
  else
    plot(x, y, main="A plot from x, y!")
}
```

Localizing Can Be Easy or Hard

```
my.plotter(my.list=list(x=-10:10, y=(-10:10)^3))
```



Localizing Can Be Easy or Hard

```
# Easy to understand error message
my.plotter()
```

```
## Error in my.plotter(): argument "x" is missing, with no default
```

Localizing Can Be Easy or Hard

```
# Not as clear
my.plotter(my.list=list(x=-10:10, Y=(-10:10)^3))
```

```
## Error in xy.coords(x, y, xlabel, ylabel, log): 'x' is a list, but does not have components 'x' and 'y
```

Who called `xy.coords()`? (Not us, at least not explicitly!)

Why is it saying `'x'` is a list? (We never set it to be so!)

`traceback()`

Calling `traceback()`, after an error: traces back through all the function calls leading to the error

- Start your attention at the "bottom", where you recognize the function you called.
- Read your way up to the "top", which is the lowest-level function that produces the error.
- Often the most useful bit is somewhere in the middle.

traceback()

If you run `my.plotter(my.list=list(x=-10:10, Y=(-10:10)^3))` in the console, then call `traceback()`, you'll see:

```
> traceback()
5: stop("'x' is a list, but does not have components 'x' and 'y'")
4: xy.coords(x, y, xlabel, ylabel, log)
3: plot.default(my.list, main = "A plot from my.list!")
2: plot(my.list, main = "A plot from my.list!") at #4
1: my.plotter(my.list = list(x = -10:10, Y = (-10:10)^3))
```

We can see that `my.plotter()` is calling `plot()` is calling `plot.default()` is calling `xy.coords()`, and this last function is throwing the error

Why? Its first argument `x` is being set to `my.list`, which is OK, but then it's expecting this list to have components named `x` and `y` (ours are named `x` and `Y`)

Debugging Tools

`cat()`, `print()`

Most primitive strategy: manually call `cat()` or `print()` at various points, to print out the state of variables, to help you localize the error.

This is a "stone knives and bear skins" approach to debugging that is still very popular among some people.

An actual quote from Stack Overflow:

> I've been a software developer for over twenty years [...] I've never had a problem I could not debug using some careful thought, and well-placed debugging print statements. Many people say that my techniques are primitive, and using a real debugger in an IDE is much better. Yet from my observation, IDE users don't appear to debug faster or more successfully than I can, using my stone knives and bear skins.

Specialized Tools for Debugging

R provides you with many debugging tools.

Why should we use them, and move past our handy `cat()` or `print()` statements?

Let's see what our primitive hunter found on Stack Overflow, after a receiving bunch of suggestions in response to his quote:

> Sweet! [...] Very illuminating. Debuggers can help me do ad hoc inspection or alteration of variables, code, or any other aspect of the runtime environment, whereas manual debugging requires me to stop, edit, and re-execute.

browser()

One of the simplest but most powerful built-in debugging tools in R is `browser()`.

You place a call to `browser()` at any point in your function that you want to debug.

```
my.fun = function(arg1, arg2, arg3) {
  # Some initial code
  browser()
  # Some final code
}
```

Then redefine the function in the console, and run it.

Once execution gets to the line with `browser()`, you'll enter an interactive debugging mode.

Things to Do While Browsing

While in the interactive debug, you can:

- type any normal R code into the console, to be executed within in the function environment.
- so you can, for example, investigate the values of variables defined in the function.

You can also type:

- `n` (or simply return) to execute the next command
- `s` to step into the next function
- `f` to finish the current loop or function
- `c` to continue execution normally
- `Q` to stop the function and return to the console

To print any variables named `n`, `s`, `f`, `c`, or `Q`, defined in the function environment, use `print(n)`, `print(s)`, etc.

Browsing in RStudio

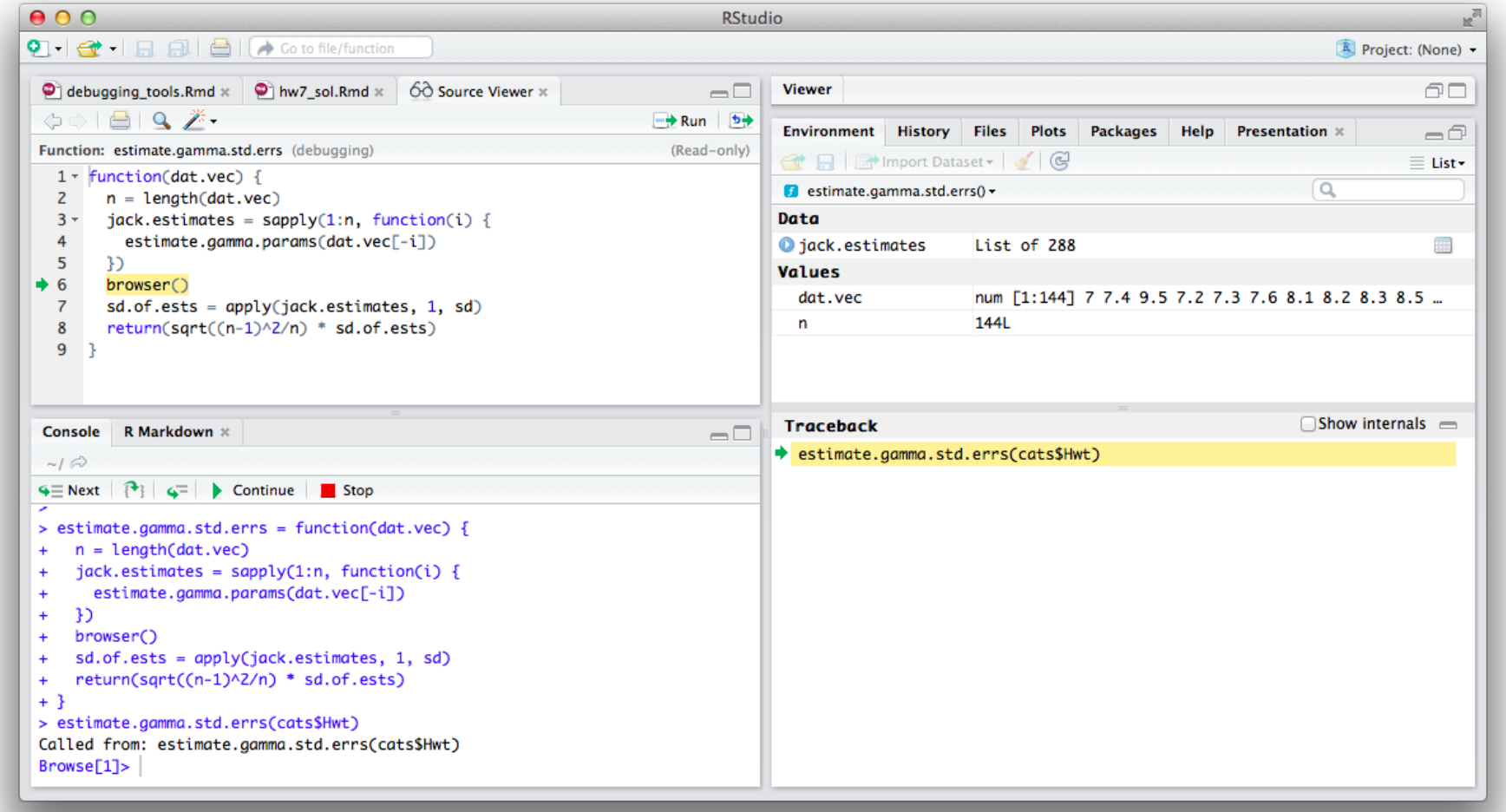
You can also control the browsing using RStudio.

- This is part of what makes RStudio an Integrated Development Environment (IDE) for R.

This gives you:

- buttons to click to navigate the code.
- access to local variables in the "Environment" panel.
- access to the traceback chain in the "Traceback" panel.

Browsing in RStudio



RStudio Debugging with `my.plotter()`

Add `browser()` inside the `my.plotter()` function definition.

```
my.plotter = function(x, y, my.list=NULL) {
  browser()
  if (!is.null(my.list))
    plot(my.list, main="A plot from my.list!")
  else
    plot(x, y, main="A plot from x, y!")
}
```

Call `my.plotter()` with the problematic input.

```
my.plotter(my.list=list(x=-10:10, Y=(-10:10)^3))
```

Knitting and Debugging

You should only run `browser()` in the console, never in an R Markdown code chunk that is supposed to be evaluated when ultimately knitting.

But, to keep track of your debugging code (that you'll run in the console), you can still use code chunks in R Markdown, you just have to specify `eval=FALSE` in the header for the code chunk.

```
# As an example, here's a code chunk that we can keep around in this Rmd doc,
# but that will never be evaluated (because eval=FALSE) in the Rmd file.

big.mat = matrix(rnorm(1000)^3, 1000, 1000)
big.mat

# Note that the output of big.mat is not printed to the console, and also
# that big.mat was never actually created! (This code was not evaluated.)
```

Examples

—

The Return of Steepest Descent

Various Flavors of a Steepest Descent Function

Recall: To find a local minimum of a function $f(x)$, take steps according to **steepest descent**:

$$x_n = x_{n-1} - \gamma f'(x_{n-1})$$

until $|f'(x_n)|$ is sufficiently small.

Sketch:

Flavor 1

```r
#' Approximate the local minimum of a function f using its derivative df.
#' @param f A function to be minimized.
#' @param df The derivative of the function.
#' @param x0 An initial guess at the minimum.
#' ... (Rest of function documentation goes here.)
steepest.descent <- function(f, df, x0,
                             max.its = 1000,
                             step.size = 0.01,
                             deriv.tol = 1e-3){

  x.new <- x0
  df.new <- f(x.new)

  num.its <- 0

  while ((df.new > deriv.tol) && (num.its < max.its)){
    x.old <- x.new

    x.new <- x.old - step.size*df.new
    df.new <- f(x.new)

    num.its <- num.its + 1
  }

  return(list(x.final = x.new, f.final = f(x.new), df.final = df(x.new), num.it
}
```

Testing a Function: Start with an Input Where You Know the Answer

As a rule, you should test a function with an input that you know will give the right answer.

For example, we know that the global minimum of $f(x) = x^2$ occurs at $x = 0$.

```
steepest.descent(function(x) x^2, function(x) 2*x, 1)
```

```
## $x.final
## [1] 0.09071079
##
## $f.final
## [1] 0.008228448
##
## $df.final
## [1] 0.1814216
##
## $num.its
## [1] 1000
```

Something went wrong, but what? Use `browser()`.

Flavor 2

```
steepest.descent <- function(f, df, x0,
                             max.its = 1000,
                             step.size = 0.01,
                             deriv.tol = 1e-3){

  x.new <- x0
  df.new <- df(x.new)

  num.its <- 0

  while ((df.new > deriv.tol) && (num.its < max.its)){
    x.old <- x.new

    x.new <- x.old - step.size*df.new
    df.new <- df(x.new)

    num.its <- num.its + 1
  }

  return(list(x.final = x.new, f.final = f(x.new), df.final = df(x.new), num.it
}
```

Flavor 2 Tests

```
steepest.descent(function(x) x^2, function(x) 2*x, 1)
```

```
## $x.final
## [1] 0.0004923008
##
## $f.final
## [1] 2.423601e-07
##
## $df.final
## [1] 0.0009846016
##
## $num.its
## [1] 377
```

Okay, that fixed $x_0 = 1$, but what about another case?

Flavor 2 Tests

```
steepest.descent(function(x) x^2, function(x) 2*x, -2)
```

```
## $x.final
## [1] -2
##
## $f.final
## [1] 4
##
## $df.final
## [1] -4
##
## $num.its
## [1] 0
```

Now something **else** went wrong. Again, use `browser()`.

Flavor 3

```
steepest.descent <- function(f, df, x0,
                             max.its = 1000,
                             step.size = 0.01,
                             deriv.tol = 1e-3){
  x.new <- x0
  df.new <- df(x.new)

  num.its <- 0

  while ((abs(df.new) > deriv.tol) && (num.its < max.its)){
    x.old <- x.new

    x.new <- x.old - step.size*df(x.old)
    df.new <- df(x.new)

    num.its <- num.its + 1
  }

  return(list(x.final = x.new, f.final = f(x.new), df.final = df(x.new), num.it
}
```

Flavor 3 Tests

```
steepest.descent(function(x) x^2, function(x) 2*x, 1)
```

```
## $x.final
## [1] 0.0004923008
##
## $f.final
## [1] 2.423601e-07
##
## $df.final
## [1] 0.0009846016
##
## $num.its
## [1] 377
```

Flavor 3 Tests

```
steepest.descent(function(x) x^2, function(x) 2*x, -2)
```

```
## $x.final
## [1] -0.0004953899
##
## $f.final
## [1] 2.454111e-07
##
## $df.final
## [1] -0.0009907797
##
## $num.its
## [1] 411
```

Are We Done?

We may be fairly satisfied that we have the code working to minimize $f(x) = x^2$.

But what about all the other functions we might want to minimize?

We can't test the code for **all possible functions**, but it would be a good idea to try at least a few more.

Mini-Exercise: Test the final version of `steepest.descent()` using at least one other function, and convince yourself that `steepest.descent()` works as-advertised.

Are We Done?

We also haven't really checked that `deriv.tol` and `max.its` are working as advertised.

Mini-Exercise: Change `deriv.tol` and `max.its` and verify that the output changes to match the new tolerance on the derivative and the maximum number of iterations we will take.

Exercises

Exercise 1

```r
library(MASS)
data(cats)

#' Compute summary statistics excluding a given sex of cat
#' from the cats data frame.
#' @param sex The sex to exclude from the data frame.
#' @return Summary statistics for the data frame excluding
#'         cats with the passed sex.
cats.stats.exclude.sex <- function(sex = "F"){
  cats.use <- cats[-(cats$Sex == sex), ]

  return(summary(cats.use))
}
```

Why do the following two function calls give the same answer?

```r
summary(cats)

cats.stats.exclude.sex(sex = "F")
```

Exercise 2

```
#' Returns the roots of a quadratic polynomial given in the form
#' ax^2 + bx + c.
#' @param a the coefficient on x^2
#' @param b the coefficient on x
#' @param c the coefficient on 1
#' @return a vector containing the roots of the polynomial.
quadratic.solver <- function(a,b,c) {
  determinant <- b^2 - 4*a*c

  return(c((-b+sqrt(determinant))/2*a, (-b-sqrt(determinant))/2*a))
}
```

Why does this work:

```
quadratic.solver(1, 0, -1)
```

but this does not work?

```
quadratic.solver(1, 0, 1)
```

Exercise 3

Recall that a generic sine wave as a function of time t (in seconds) with amplitude A and temporal frequency f (in Hz) is

$$s(t) = A\sin(2\pi f t).$$

A sine wave with frequency f will repeat f times per second.

`plot.sine()` plots a sine wave with a provided frequency and amplitude.

```
plot.sine <- function(frequency, amplitude, to = 1){
  x <- seq(0, to, length.out = 1000)
  y <- amplitude*sin(2*pi*frequency*x)

  plot(x, y, type = 'l')
}
```

Why does this work:

```
plot.sine(2, 5)
```

while this does not?

```
plot.sine(amplitude <- 5, frequency <- 2)
```